8

Árvores de Busca Binária

" Testes experimentais de programas podem, quando muito mostrar a presença de erros, mas nunca podem mostrar a sua ausência "

- Niklaus Wirth -

Sumário:

8.1- Introdução

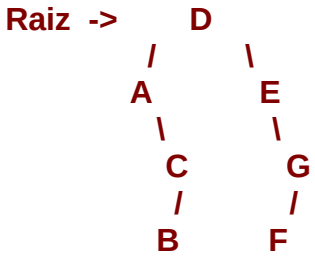
No capítulo anterior estudamos estruturas de dados não lineares, as àrvores binárias. Quando os campos dessa estrutura apresentam uma relação de ordem entre as suas chaves, essas árvores binárias são chamadas de árvores de busca binária e, são muito utilizadas para extração de informação nos sistemas geradores de bases de dados.

Em toda a arvores de busca binária as operações de consulta são muita rápidas quando estas estão equilibradas, ou seja, a sua subárvore esquerda tem aproximadamente a mesma profundidade do que a subárvore à direita.

8.2- Definições e Propriedades

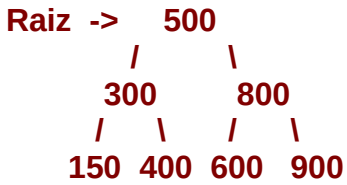
Uma árvore busca binária T é uma árvore de busca binária que satisfaz as seguintes propriedades:
- T é vazia;
- O conteúdo das subárvores à esquerda são menores do que a raiz;
- O conteúdo das subárvores à direita são maiores do que a raiz;
- As subárvores à esquerda e à direita são árvores de busca binária.

Consideremos por exemplo a árvore de busca binária.



Se percorrermos essa árvore em in-ordem e considerarmos que a operação visitatomo consiste na impressão do conteudo da chave do átomo, obteremos a seguinte sequencia ordenada: A, B, C, D, E, F, G.

Vejamos um outro exemplo:



Se percorremos está árvore em in-ordem teremos a seguinte sequência de númerica ordenada: 150, 300, 400, 500, 600, 800, 900

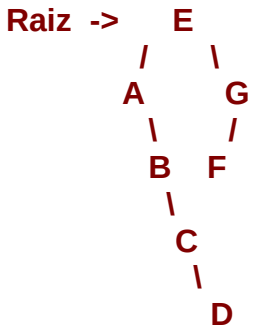
Para obtermos uma sequência de caracteres em ordem decrescente basta invertermos as propriedades da árvores de busca binária.

8.3 - Construção de uma Árvore de Busca Binária

Uma árvore de busca binária pode ser construída a partir de um conjunto finito de dados. Consideremos sem perda da generalidade o conjunto.

{ E, A, B, G, F, C, D }

Se percorrermos esse conjunto da esquerda para a direita e aplicarmos a definição de uma árvore de busca binária, obteremos:



8.4 - Tipo Abstracto de Dados

Um possível conjunto de operações para criar um tipo abstracto de dados árvore de busca binária é descrito pelas operações:

Consultar uma Chave;
Inserir um Elemento;
Remover um Elemento

Vimos que o armazenamento de árvores pode ser feito em organização sequencial ou dinâmica e que devido a devido a natureza recursiva desta estrutura os algoritmos de manipulação ficam mais fáceis com a organização dinâmica.

8.4.1- Códigos de Erro

```
#define NOT_FOUND      -1        // Item não existe
#define  OK             0        // Operação realizada com sucesso
#define  TREE_EMPTY     2        //  Arvore vazia
#define NO_SPACE        5       // Não há espaço de memória
```

8.4.2- Estrutura de Dados

Vimos que o armazenamento de árvores pode ser feito em organização sequencial ou dinâmica e que devido a natureza recursiva desta estrutura faremos apenas a implementação em organização dinâmica.

8.4.1- Códigos de Erro

```
#define NOT_FOUND      -1          // Item não existe
#define  OK             0          // Operação realizada com sucesso
#define  TREE_EMPTY     2          //  Arvore vazia
#define NO_SPACE        5         // Não há espaço de memória
```

8.4.2- Estrutura de Dados

```
typedef struct
 {
   int   chave;
   int   valor;
 }TInfo;

typedef struct Atomo
 {

  TInfo   info;
  struct Atomo *fesq;
  struct Atomo *fdir;
  struct Atomo *pai;
 }TAtomo;
```

Declaramos uma árvore binária em organização dinâmica com um ponteiro raiz que faz referencia ao primeiro átomo da árvore.

```
typedef struct
  {
      TAtomo * raiz;
   } TAB;

typedey  enum   {FALSE= 0, TRUE= 1} Boolean;
```

8.4.3- Implementação das Operações

A operação para encontrar um átomo numa árvore de busca binária começa pela raiz da árvore e, consiste em comparar o valor do átomo que queremos encontrar com a chave. Se o valor for maior, devemos efectuar um processo de busca pela subárvore à direita, desprezando deste modo, toda a informação que está armazenada na subárvore à esquerda. Mas se o valor for menor, devemos efectuar o processo de busca pela subárvore à esquerda e como

consequência, toda a informação que está armazenada na subárvore à direita será desprezada. Este algoritmo leva-nos a implementação da seguinte função:

```
*/-------------------------------------------------------------------------------------------------
Recebe:  Arvore de busca binária,  conteudo da chave de busca
Objectivo: Encontrar um átomo que possui essa chave
Devolve: NULL se o elemento não existe ou o endereço do átomo
------------------------------------------------------------------------------------------------- */
TAtomo  * buscaABB (TAB *arvore,  int  x)
{
  TAtomo *pbusca = arvore->raiz;
  while ( pbusca != NULL )
    {
       if ( pbusca->info.chave == x )
          return pbusca;
       else if ( pbusca->info.chave > x )
                 pbusca = pbusca->fesq;
            else
                 pbusca = pbusca->fdir;
     }
   return NULL;
}
```

cuja versão recursiva é descrita a seguir:

```
TAtomo  * buscaABBRec (TAB *arvore,  int  x, TAtomo *pbusca)
{
  if ( pbusca  == NULL )
      return NULL;
  if ( pbusca->info.chave == x )
     return pbusca;
  if ( pbusca->info.chave > x )
      return buscaABBRec ( arvore, x, pbusca->fesq );
  else
      return buscaABBRec (arvore, x, pbusca->fdir);
 }
```

A operação para inserir um novo átomo numa árvore de busca binária deve ser feita de tal forma que a árvore resultante deve continuar a satisfazer as propriedades de uma árvore de busca binária. Uma forma de garantir essa propriedade é inserir os átomos como folhas.

Mas, para fazer essa inserção necessitamos de descrever uma estratégia para percorrermos à árvore da raiz até ao átomo que será o pai da folha que iremos inserir.

Este percurso começa na raiz. Em cada bifurcação, comparamos o valor que pretendemos inserir com o conteúdo do átomo esquerdo e direito. Seleccionamos à bifurcação que satisfaça a definição da árvore de busca

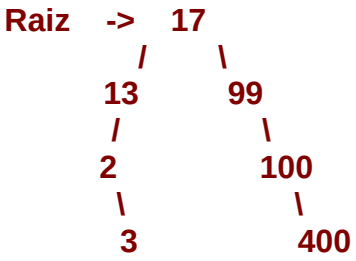
binária e baixamos no nível da árvore até chegarmos à uma folha. Inserimos o novo átomo como descendente dessa folha.

Vejamos uma simulação deste algoritmo para o seguinte conjunto de dados:

{17,99,13,1,3,100,400}

Vamos varrer o conjunto da esquerda para a direita. Inicialmente à árvore está vazia. Inserimos o elemento 17 como raiz da árvore. Nesta altura criamos uma árvore com um único átomo. Em seguida, lemos o elemento 99 e comparamos esse elemento com a raiz. Como 99 é maior do que 17, inserimos esse elemento na subárvore direita. Mas à árvore é unitária, logo 99 é o filho direito da raiz. Para inserir o terceiro elemento do conjunto, o número 13, compramos esse elemento com a raiz. Como 13 é menor do que a raiz, inserirmos esse elemento na subárvore esquerda. Mas à raiz não possui subárvore esquerda, então 13 é o filho esquerdo da raiz. Para inserirmos o quarto elemento, o número 1, comparamos esse elemento com a raiz. Como 1 é menor do que o conteúdo da raiz, baixamos de nível pela subárvore esquerda. Em seguida, comparamos o valor desse elemento ao conteúdo do filho esquerdo da raiz. Mas o filho esquerdo da raiz é uma folha, logo esse elemento é inserido como filho esquerdo dessa folha. O próximo elemento a ser inserido é o número 3. Como 3 é menor do que raiz, baixamos de nível pela subárvore esquerda. Comparamos em seguida esse elemento com o número 13. Como o elemento que pretendemos inserir é menor, baixamos mais uma vez de nível pela subárvore esquerda e compráramos o elemento com o número 1. Como o número 1 é uma folha e o elemento que pretendemos inserir é maior, inserimos o como filho direito dessa folha.

Repita a titulo de exercício este procedimento para os restantes números até obtermos à seguinte árvore:



Agora, estamos em condições de desenvolver um algoritmo que implementa essa estratégia.

O algoritmo começa por solicitar um átomo à memória. Se a operação for bem sucedida preencher esse átomo com a informação que queremos inserir e, verificar em seguida se árvore está vazia. Se essa condição for verdadeira, formar uma árvore unitária. No caso contrário, percorrer a árvore até chegarmos à uma folha. Esse percurso começa pela raiz e consiste em comparar a chave do átomo que pretendemos inserir com a chave da raiz. Se o valor da chave da árvore for menor devemos efectuar o processo de busca

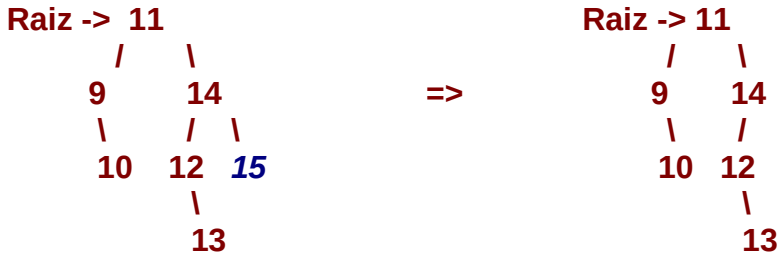
pela subárvore á direita; Se for maior ou igual, efectuar o processo de busca pela subárvore à esquerda.

```
*/-------------------------------------------------------------------------------------------------------
Objectivo: Inserir um elemento
Parâmetro Entrada: árvore de busca binária, Informação a inserir
Parâmetro de Saída: árvore binaria actualizada
Valor de Retorno: Código de Erro (NO_SPACE ou OK)
------------------------------------------------------------------------------------------------------- */
int  inserirABB (TAB *arvore, Tinfo  x, )
{
   TAtomo *pnovo = criarFolha(x);
    if ( pnovo == NULL )
        return NO_SPACE;
    if ( arvore->raiz == NULL )                          \\  Árvore Vazia
      {
        pnovo->pai = NULL;
        arvore->raiz = pnovo;
      }
    else
      {
          TAtomo  *ppai= NULL, * paux = arvore->raiz;  \\ Procurar  o lugar
          while ( paux!= NULL )
           {
              ppai = paux;
              if ( pnovo->info.chave > paux->info.chave )
                 paux = paux->fdir;
              else
                  paux = paux->fesq;
           }
          pnovo->pai = ppai;                                  \\ Ligar o átomo
          if ( ppai->info.chave > pnovo->info.chave )
             ppai->fesp = pnovo;
          else
              ppai->fdir = pnovo;
      }
     return ok;
}
```

A operação para remoção um átomo numa árvore de busca binária é muito complexa porque muitas vezes temos de reorganizar de uma parte da árvore. Essa reorganização deve ser feita de modo a preservar as propriedades de uma árvore de busca binária.

Seja x um átomo qualquer de uma árvore de busca binária. Para remove-lo, devemos de considerar três casos:

1º Caso: se x não possui filhos a remoção consiste em cortar a ligação desse átomo a árvore. Veja o caso do átomo com conteúdo 15.



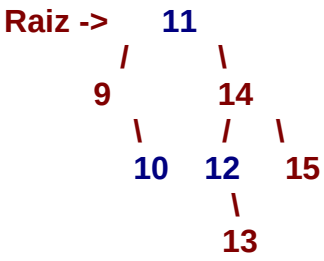
2º Caso: Se x possui apenas um filho a remoção consiste em ligar o pai de x ao filho de x, em outros termos, ligar o pai de x ao seu neto. Veja o caso do átomo com conteúdo 5.

3º Caso: Se x possui dois filhos. Para esta situação temos duas casos a considerar:

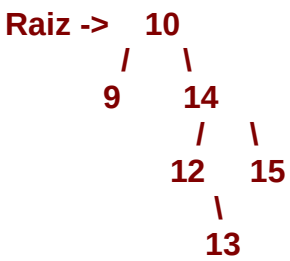
- Substituir o átomo x pelo maior valor da subárvore à esquerda. Este valor encontra-se no átomo mais à direita da subárvore a esquerda.

- Substituir o átomo x pelo menor valor da subárvore à direita. Este valor encontra-se no átomo mais à esquerda da subárvore à direita.

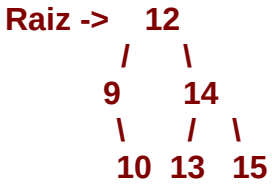
Vejamos um exemplo ilustrativo. Suponhamos que pretendemos remover a raiz da árvore, temos duas hipóteses



A primeira consiste em substituir o átomo com o conteúdo 10 para a raiz. Essa movimentação produzirá a seguinte árvore.



enquanto a segunda consiste em substituir a raiz pelo átomo cujo conteúdo é igual a 12. Para esse caso, o átomo com o conteúdo igual á 13 deverá ocupar o lugar do átomo que foi movimentado para a raiz.



A consequência directa destas propriedades é que a menor chave está sempre armazenada no átomo localizado na extremidade mais à esquerda da árvore. Isso leva-nos de definir uma função miniABB() que recebe como parâmetro de entrada uma árvore de busca binária e devolve via valor de retorno o endereço do átomo que contêm essa chave.

```
TAtomo *miniABB (TAB  *arvore)
 {
    if ( arvore == NULL ) return NULL ;
    if (arvore -> fesq == NULL ) return arvore ;
    return  miniABB (arvore->fesq);
  }
```

Para movimentar um átomo entre dois endereços de uma árvore de busca binária, utilizaremos a função que descrevemos a seguir.

```
void moverAtomo ( TAB *arvore, TAtomo * destino, TAtomo * origem )
{
  if ( destino != NULL )
    {
      if ( destino-> pai == NULL )                // destino é a raiz da ABB
         arvore = origem ;
      else
        {
          if ( destino == destino -> pai -> esq )
              destino -> pai -> esq = origem ;
          else
             destino -> pai -> dir = origem ;
          if ( origem != NULL )
               origem -> pai = destino -> pai ;
        }
    }
}
```

Com base nessas funções auxiliares, vamos implementar uma função que recebe como parâmetro de entrada uma árvore de busca binária e uma determinada chave. Remover essa chave e reorganizar a árvore de tal forma que ela continua a preservar as propriedades iniciais.

```
*/-------------------------------------------------------------------------------------------------
Objectivo: Remover um átomo dado uma chave.
Parâmetro Entrada: Arvore e a chave
Parâmetro Saída: árvore binaria actualizada.
Valor de Retorno: Código de erro (NOT_FOUND ou OK)
------------------------------------------------------------------------------------------------- */
int  removerABB (TAB *arvore,  int  x)
{
  TAtomo * paux1 = buscaABB(&arvore,x );
   if ( paux1 == NULL )
      return NOT_FOUND;
   if (paux1 -> fesq == NULL)                                  /* Caso 1 ou 2 */
      moverAtomo(&arvore , paux1 , paux1- >fdir );
  else
     {
       if (paux1->fdir == NULL)                                /* Caso 2 */
          moverAtomo(&arvore, paux1,paux1->fesq );
       else {
              TAtmo  * paux2 = miniABB(paux1 -> fdir );        /* Caso 3 */
               if (paux2->pai != paux1 )
                 {
                   moverAtomo (&arvore,paux2 , paux2->fdir );
                   paux2->fdir = paux1->fdir;
                   paux2->fdir ->pai = paux1;
                 }
              moverAtomo(&arvore , paux1 , paux2);
              paux2 ->fesq = paux1- > fesq ;
              paux2->fesq -> pai = paux1 ;
            }
      }
    free ( paux1);
    return OK;
  }
```

8.5- Exercícios

8.5.1- Dado os conjuntos :

{ 32, 8, 4, 16, 6, 64, 48, 7, 128 }

{ 16, 7, 8, 64, 6, 48, 128, 4, 32}

Construa as correspondentes árvores de busca binária.

8.5.2- Desenvolva uma função recursiva que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore binária. Verificar se ele é ou não uma árvore de busca binária.

8.5.3- Desenvolva uma função recursiva que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore de busca binária. Devolver como valor de retorno o elemento que contêm o maior conteúdo na subárvore a esquerda.

8.5.4- Desenvolva uma função iterativa que recebe como parâmetro de entrada uma lista ligada e devolve como parâmetro de saída a correspondente árvore de busca binária.

8.5.5- Desenvolva uma função iterativa que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore de busca binária e o endereço de um determinado átomo. Devolver o endereço do átomo antecessor.

8.5.6- Desenvolva uma função recursiva que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore de busca binária e uma determinada chave. Devolver o nível do átomo que contêm essa chave.

8.5.7- Desenvolva uma função iterativa que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore de busca binária e dois valores. Retornar o endereço do ancestral comum mais próximo desses valores. Caso algum dos valores não esteja na árvore retornar o valor NULL.

8.5.8- Desenvolva uma função que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore de busca binária e um índice i entre 0 e n-1, onde n é o número de elementos da árvore, e devolva o i-ésimo menor elemento da árvore.

8.5.9- Desenvolva uma função que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore de busca binária, construir uma árvore de busca binária de com fios. Considere que a estrutura dos átomos dessa árvore contêm os ponteiros sucessor e antecessor e que estão inicializados com NULL.

8.5.10- Desenvolva um procedimento recursivo que recebe como parâmetro de entrada um ponteiro para uma árvore de busca binária . Imprimir o conteúdo das chaves das folhas em ordem crescente.

8.5.11- Desenvolva uma função que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore e busca binária e uma determinada chave. Remover o átomo que contêm essa chave se for uma folha.

8.5.12- Desenvolva uma função recursiva que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore de busca binária e uma determinada informação. Inserir essa informação, mas suponha que nessa árvore não podemos ter chaves repetidas.

8.5.13- Desenvolva uma função que recebe como parâmetro de entrada um ponteiro para a raiz de uma árvore e busca binária e uma determinada chave. Remover o átomo que contêm essa chave se ele possuir um e apenas um filho.